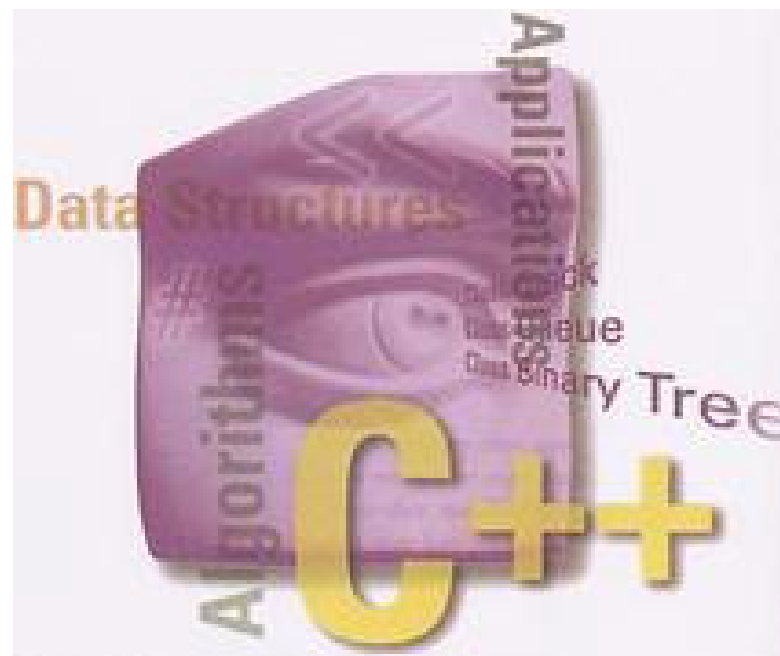


資料結構(Data Structures)

Course 5: Stack and Queue

授課教師：陳士杰

國立聯合大學 資訊管理學系





● 本章重點

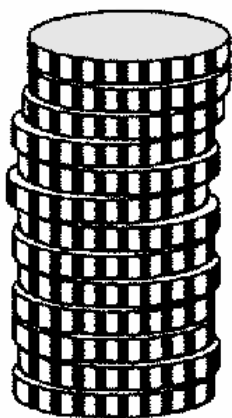
- ❖ Stack的定義、應用、製作與ADT
- ❖ Queue的定義、應用、製作與ADT
- ❖ 如何利用Array與Linked list製作Stack與Queue
- ❖ Infix(中序)運算式與Postfix (後序), Prefix (前序) 運算式間之相互轉換
- ❖ Postfix與Prefix的計算 (Evaluation)
- ❖ Stack Permutation



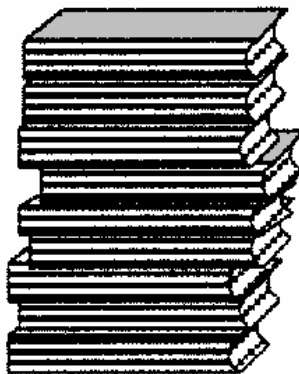
Stack (堆疊)

● Def: 具有LIFO (last in-first out)或FILO (first in-last out)性質的有序串列。

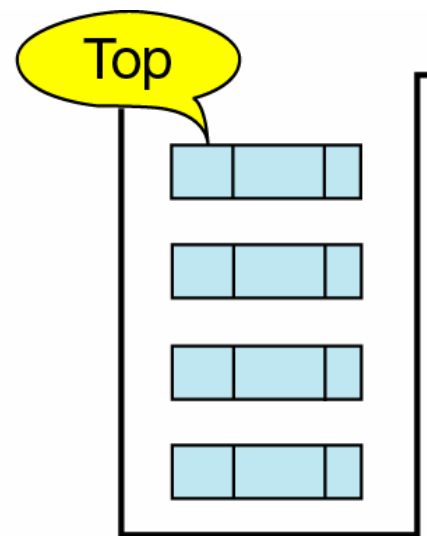
- ✦ 插入元素的動作稱為**Push**, 刪除元素的動作稱為**Pop**.
- ✦ Push/Pop的動作皆發生在同一端, 此端稱為**Top**.



Stack of coins



Stack of books



Computer stack



Stack之ADT

● Data Object Spec.

- ✦ A **set** of data item
- ✦ **Top**: 指出目前頂端元素所在
- ✦ **Size**: 表Stack的大小

● Operation Spec.

- ✦ Create(S)→S
 - 建立一個空的Stack S, 傳回值為一個新的Stack S, 傳回給User使用。
- ✦ Push(S, item)→S
 - 將資料item插入到Stack S中, 並成為Top端元素
 - If Stack full, 則無法執行
- ✦ Pop(S)→item, S
 - 刪除Stack S的Top端元素
 - If Stack empty, 則無法執行



❏ $\text{Top}(S) \rightarrow \text{item}$

- 傳回Stack S之Top端元素值, 但不刪除

- If Stack empty, 則無法執行

❏ $\text{IsFull}(S) \rightarrow \text{Boolean}$

- 判斷S是否為full

- 若是, 則傳回True; 否則傳回False

❏ $\text{IsEmpty}(S) \rightarrow \text{Boolean}$

- 判斷S是否為empty

- 若是, 則傳回True; 否則傳回False



※練習範例1※

- $\text{Pop}(\text{Push}(S, \text{item})) = \underline{\hspace{1cm}}$
- $\text{Top}(\text{Push}(S, \text{item})) = \underline{\hspace{1cm}}$
- $\text{IsEmpty}(\text{Create}(S)) = \underline{\hspace{1cm}}$
- $\text{Pop}(\text{Create}(s)) = \underline{\hspace{1cm}}$
- $\text{IsEmpty}(\text{Push}(S, i)) = \underline{\hspace{1cm}}$

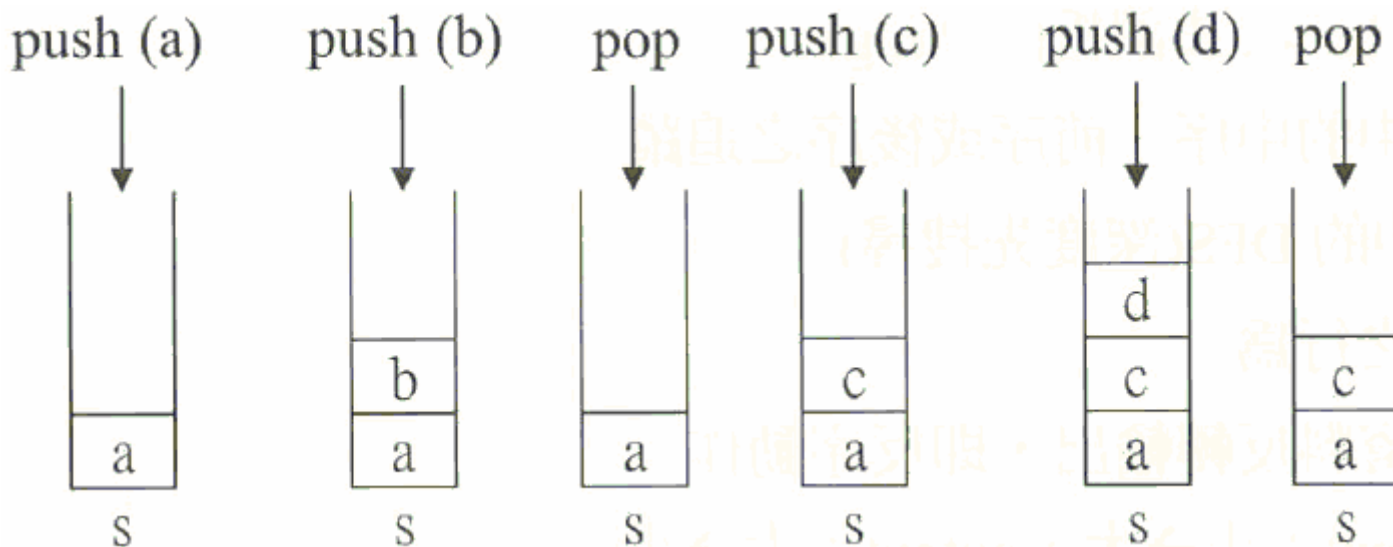


※練習範例 2※

- 有一空的Stack, 實施下列動作後, Stack的內容為何?

Push (S, a), Push(S, b), Pop(S), Push(S,c), Push(S, d), Pop(S)

Ans:





Stack 的排列組合問題(Stack Permutation)

- 三個資料a, b, c依序push入stack, 而過程中可插入pop動作, 則合法的排列組合有哪些?

Sol:

☒ abc \Rightarrow

☒ acb \Rightarrow

☒ bac \Rightarrow

☒ bca \Rightarrow

(x) ☒ cab \Rightarrow

☒ cba \Rightarrow

\therefore 共有5種合法的排列組合!!



- n個資料執行stack permutation, 其合法的排列組合個數為多少?

Sol:

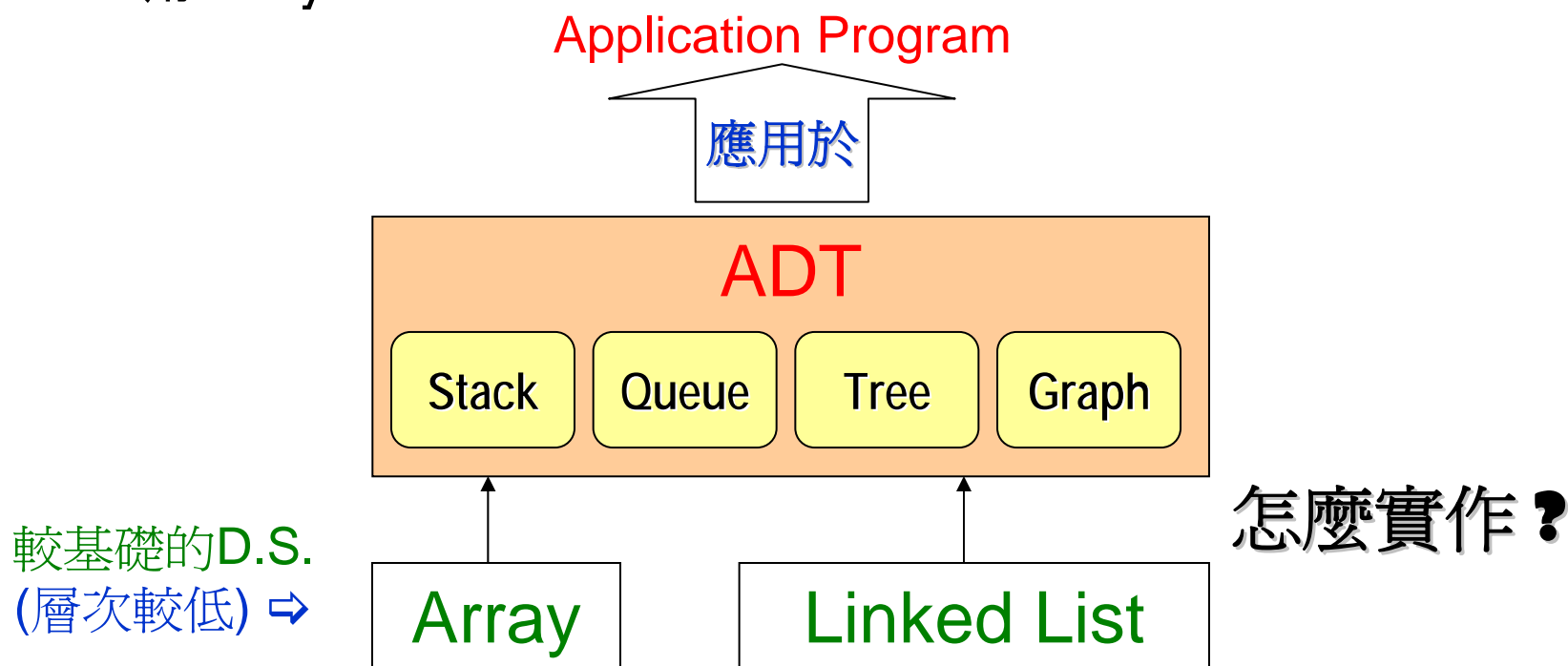
$$\frac{1}{n+1} \binom{2n}{n} \quad \leftarrow \quad \text{Catalmnn Number}$$

- Catalmnn Number可用於表示:
 - ❖ n個nodes所形成的不同二元樹個數
 - ❖ n個“(”與“)”所形成的合法配對個數
 - ❖ n個矩陣之所有可能相乘方式 (同“括號配對”的觀念)



Stack之製作

- 有兩個方式:
 - 用Linked List
 - 用Array



- 當然，我們也可以用Stack實作出Queue，或用Queue實作出Stack，但這是另一層次的問題了!!



用 Linked List 製作 Stack

- 要利用 linked list 實作 stack, 需要撰寫兩個不同的結構 (Structure):

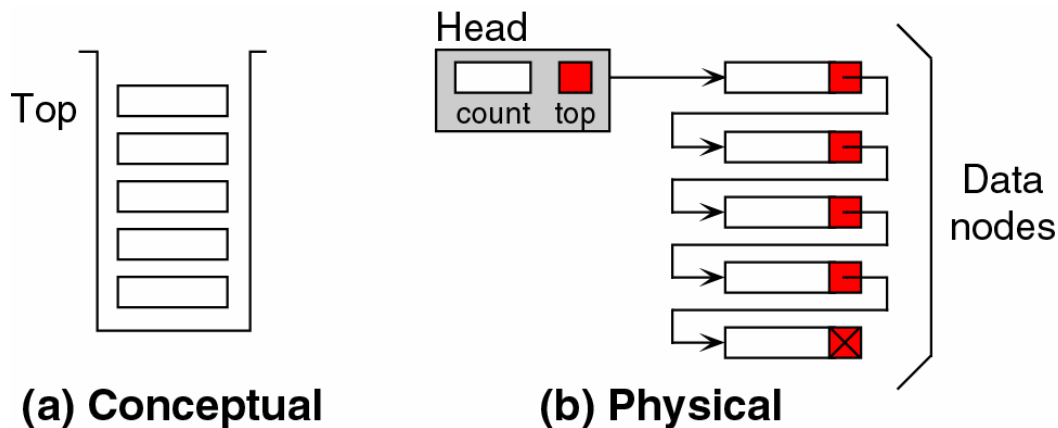
- **Head**: 用以當作堆疊中的Top, 以指出堆疊中頂端元素之所在。

- 此Head結構內不一定要有存放資料之資料變數, 但一定要有一個指標以指向堆疊的頂端元素。

- 往後各節若無特別說明, 此Head結構皆僅有一個指標, 名為“Top”。

- **Data Node**: 用以存放欲置於堆疊中的資料。

- 此節點的結構內至少要有一個指標與一個資料變數。指標用以指向下一個元素, 資料變數用以存放堆疊的資料。



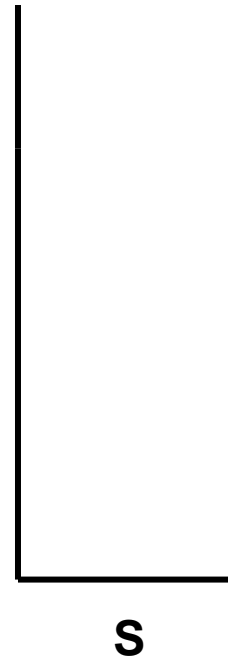


Create(S)

- 主要作法: 宣告top指標為null即可:

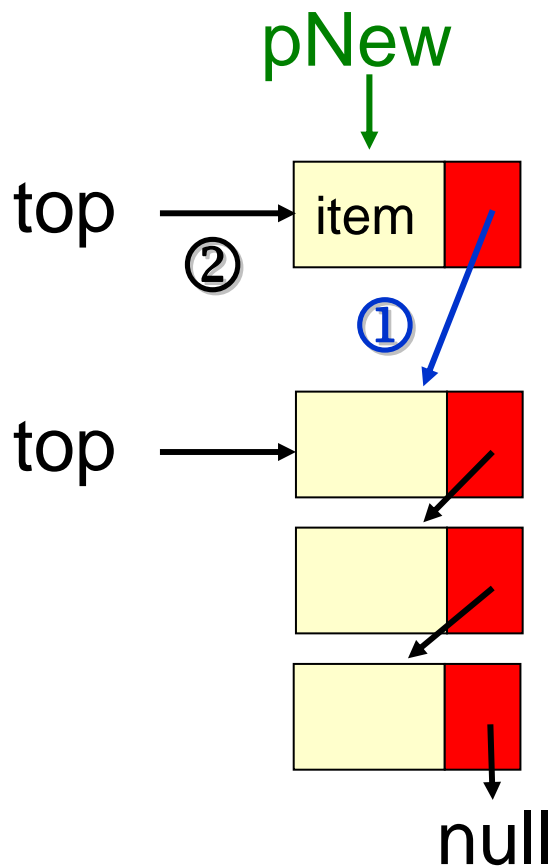
top = null (初值);

top → null

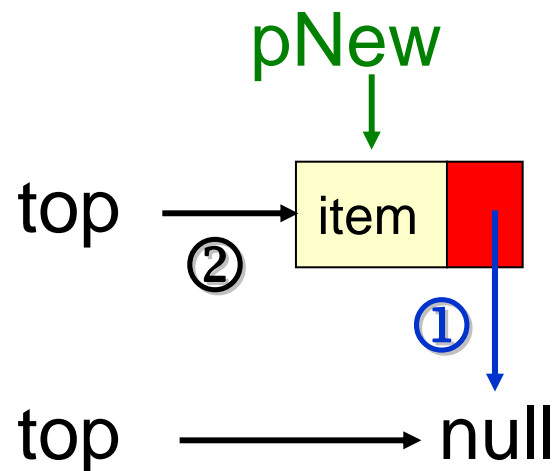


Push(S, item)

① 推入非空堆疊



② 推入空堆疊





● 主要作法:

begin

New(t); //跟系統要求記憶體空間以產生並置放一個新節點pNew

pNew→data = item; //把資料“item”塞到這個新節點pNew的Data欄位中

pNew→link = top; ①

top = pNew; ②

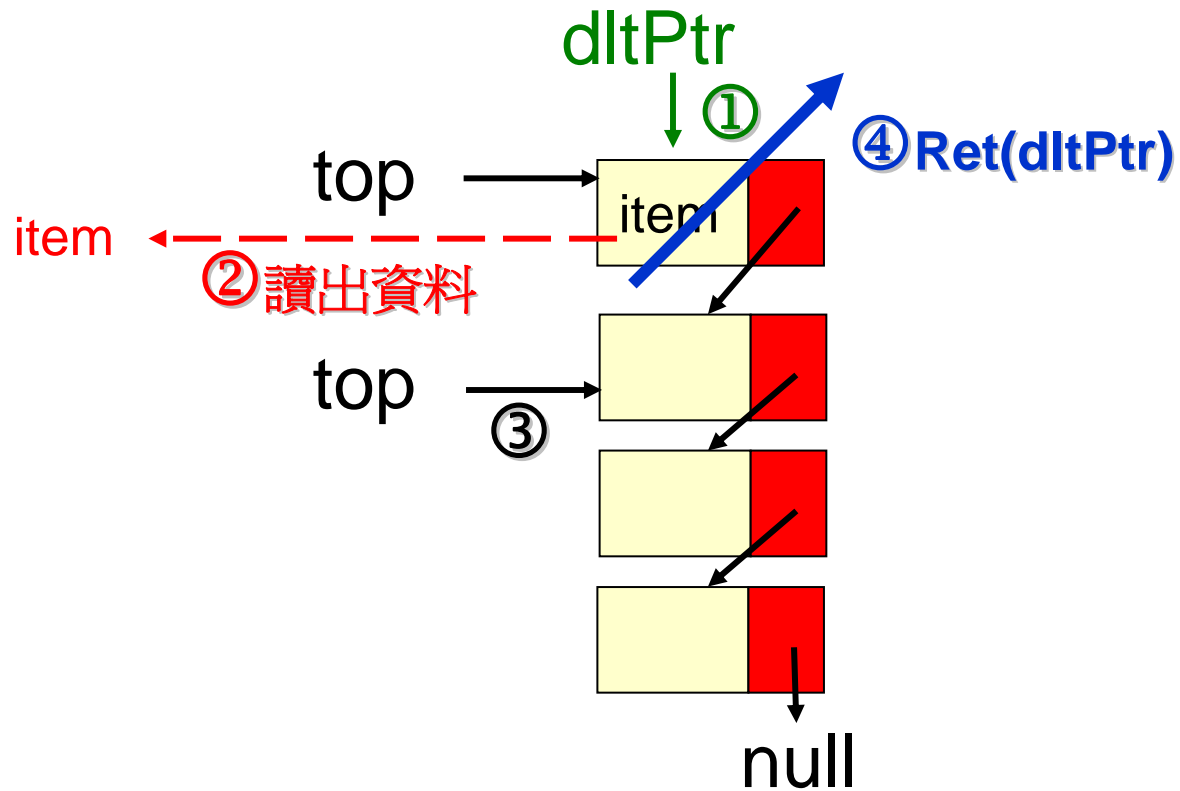
end

● 特點:

- ❏ 只要Memory有空間, O.S.就允許Linked Stack去Push新資料!!不用像Array一樣, 要先檢查Array是否滿了。



Pop(S)





● 主要作法:

begin

if (top = null)

Stack empty;

else begin

① dltPtr = top;

② item = top → data;

③ top = dltPtr → link;

④ Ret(dltPtr);

end;

end



Top(S)

● 主要作法:

```
begin
  if (top= null)
    success = false;
  else begin
    print(top→data);
    success = true;
  end;
  return success;
end
```



IsFull(S)

● 主要作法:

```
begin
  if (memory available)
    result = false;
  else
    result = true;
  return result;
end
```



IsEmpty(S)

● 主要作法:

```
begin
  if (top=null)
    result = false;
  else
    result = true;
  return result;
end
```



■ 用 Array 製作 Stack

● 我們專注於以下的 Operation Spec.

- ❏ **Create(S)** → S
- ❏ **Push(S, item)** → S
- ❏ **Pop(S)** → item, S
- ❏ **Top(S)** → item
- ❏ **IsFull(S)** → Boolean
- ❏ **IsEmpty(S)** → Boolean



Create(S)

- 建立空的Stack。

- 只需做宣告即可：

S: Array[1...n] item //宣告一個Array

int top = 0 //設定一整數變數top且初值為0

為了說明方便，以下談論用**Array**製作**Stack**時，均假設一維陣列的起始位址是從1開始!!這與實際撰寫**C++**有所出入，請特別注意。



Push(S, item)

begin

if top = n

stack Full;

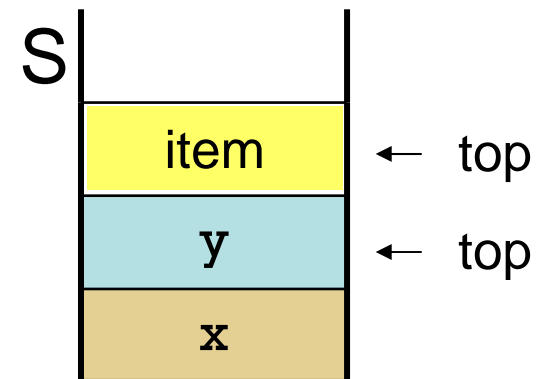
else begin

top = top + 1; //top要先加1

S[top] = item; //再將item置入

end;

end





Pop(S)

begin

if top = 0

stack empty;

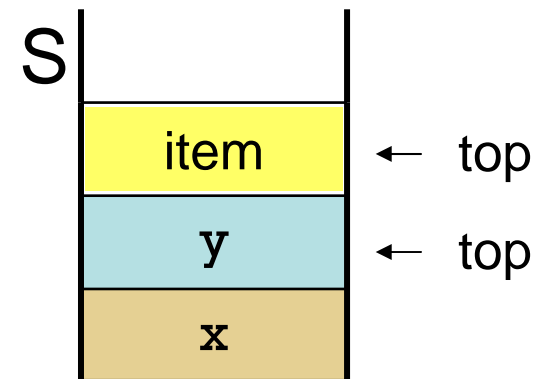
else begin

item = S[top]; //先將item叫出

top = top - 1; //再將top減1

end;

end





Top(S)

begin

if top = 0

stack empty;

else

return S[top]; //將item叫出

end



IsEmpty(S)

begin

if top = 0

return True;

else

return False;

end



IsFull(S)

begin

if top = n

return True;

else

return False;

end

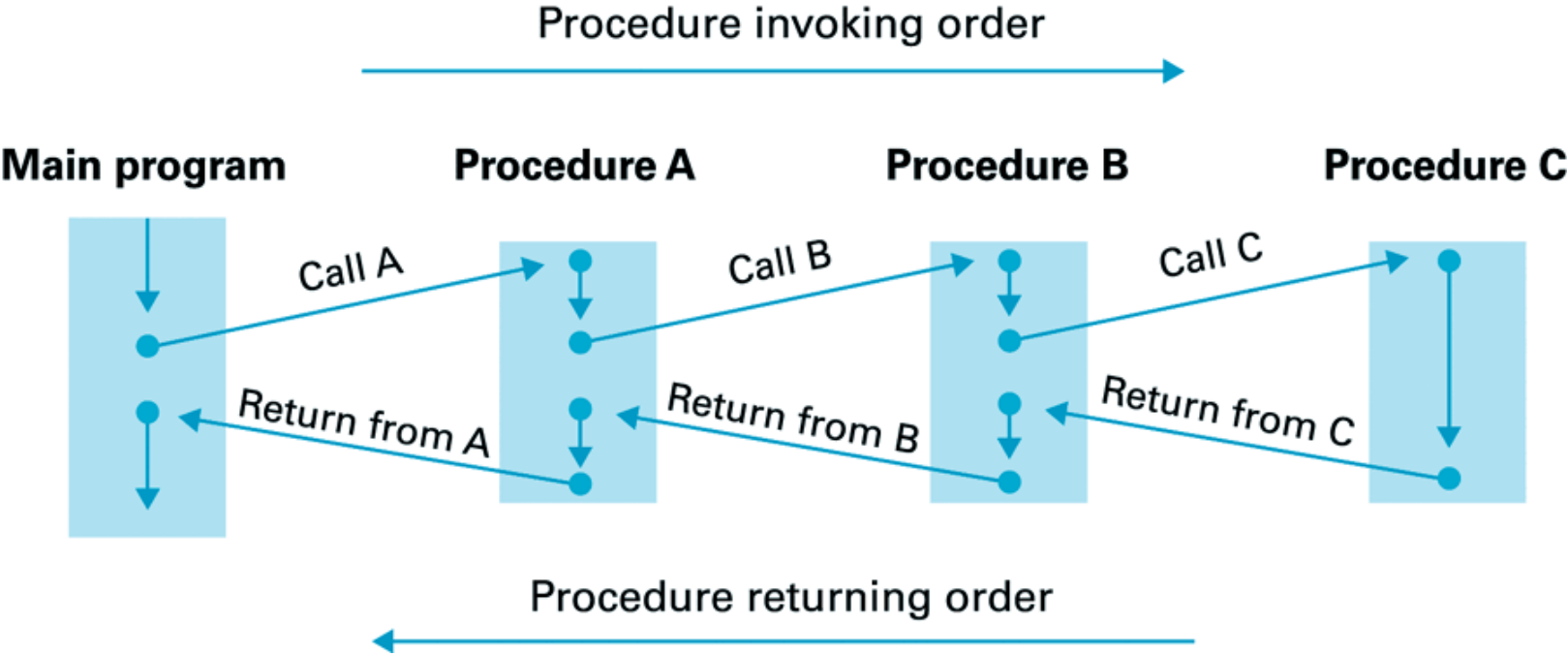


Stack Application

- Procedure Call/Recursive Call之處理
- Parsing (剖析)
- Reversing Data (反轉資料)
- 中序式 (Infix)與前序式 (Prefix)/後序式 (Postfix)間互轉
- 後序式的計算



Procedure Call/Recursive Call (副程式/遞迴呼叫)





Parsing (剖析)

- 編譯器 (Compiler) 將程式剖析成單獨的個體，如：關鍵字、名字、標誌...等，以進行程式語法檢查。
 - 一般常見的程式問題是不對稱的括號 (**Unmatched Parentheses**)，即是編譯器在剖析過程中，藉由堆疊來做判斷。

$\overbrace{((A + B) / C}^?$

(a) Opening parenthesis not matched

$? \overbrace{(A + B) / C}^?$

(b) Closing parenthesis not matched



Reversing Data (反轉資料)

放入



(a)

取出



(b)



■ Infix 與 Prefix/Postfix 間互轉

● 數學運算的表示式可有下列三種：

- ❏ Prefix (前序式): $+ab$
- ❏ Infix (中序式): $a+b$
- ❏ Postfix (後序式): $ab+$



● Infix (中序式):

- ☒ Def: 一般所使用的Expression Format

- ☒ 格式:

Operand 1 (運算元 1) Operator (運算子) Operand 2 (運算元 2)

- ☒ 運算子的種類:

- Binary: +, -, ÷, ×, and, or, ...

- Unary: not

- ☒ 缺點: 不利於Compiler對式子運算的處理

- ∴需要考慮運算子之間的優先權、結合性

- ∴Compiler可能需要來回多次scan才可以求算出結果

- Ex: $a + b * c \uparrow (d - e)$



● 結合性對 \uparrow , $-$, \div 有影響!!

❖ 左結合: $(5-3)-2 = 0$

❖ 右結合: $5-(3-2) = 4$



● Postfix (後序式):

✦ 格式:

Operand 1(運算元 1) Operand 2(運算元 2) Operator(運算子)

✦ 優點:

- Compiler易於處理, **scan一次**即可求得計算結果

- '∴'在後序式的表示式當中, 已免除掉**括號**, **優先權**與**結合性**的考量

- ✦ 中序式轉後序式, 需要用到**一個Stack**的支援。



● Prefix (前序式):

✦ 格式:

Operator(運算子) **Operand 1(運算元 1)** **Operand 2(運算元 2)**

✦ 優點:

● Compiler處理Prefix的計算, **scan一次**即可求得結果

● ∴在前序式的表示式當中, 已免除掉**括號**, **優先權**與**結合性**的考量

✦ 但是在中序式轉前序式比較麻煩, 需要用到**二個Stack**的支援。∴還是傾向使用Postfix。



● 中序式轉後序式、前序式之相關議題:

■ 中序式與後序式/前序式互轉的**計算 (Evaluation)**

- 中序式轉後序式、前序式

- 後序式、前序式轉中序式

■ 中序式與後序式/前序式互轉的**演算法 (Algorithm)**

- 中序式轉後序式的演算法

- 後序式計算的演算法



■ 中序式與後序式/前序式互轉的計算

- 中序式轉後序式、前序式
- 後序式、前序式轉中序式



中序式轉後序式、前序式

● 使用“括號法”:



❏ 中 \Rightarrow 後的步驟:

- 對於中序式, 先加上**完整的括號配對**
- 將運算子取代最近的**右**括號
- 刪除**左**括號, 予以輸出即可

❏ 中 \Rightarrow 前的步驟:

- 對於中序式, 先加上**完整的括號配對**
- 將運算子取代最近的**左**括號
- 刪除**右**括號, 予以輸出即可

一般常見的運算子之優先權與結合性：

運算子	優先權	結合性	Unary / Binary
(,)	高	左結合	
(正號) +, (負號) -	 	左結合	Unary
(冪次方) **, ↑, ^, \$		右結合	
×, ÷		左結合	
+, -		左結合	
(關係) >, <, ≥, ≤		左結合	
(邏輯) Not, ~		左結合	Unary
(邏輯) And, Or		左結合	
(指定) =		右結合	
	低		



● Ex: $A+B\times C$, 寫出其postfix及prefix

☒ 中⇒後:

● 加上完整的括號配對 (\times 優先於 $+$): $(A+(B\times C))$

● 將運算子取代最近的右括號: $(A+(B\times C)) \Rightarrow$

● 刪除左括號: $(A (BC\times + \Rightarrow \underline{ABC\times +}$

☒ 中⇒前:

● 加上完整的括號配對 (\times 優先於 $+$): $(A+(B\times C))$

● 將運算子取代最近的左括號: $(A+(B\times C)) \Rightarrow$


● 刪除右括號: $+A\times BC)) \Rightarrow \underline{+A\times BC}$

● Ex: $A+B+C$, 寫出其postfix及prefix

❏ 中⇒後:

● 加上完整的括號配對 (左結合): $((A+B)+C)$

● 將運算子取代最近的右括號: $((A+B)+C) \Rightarrow$




● 刪除左括號: $((AB+C+ \Rightarrow \underline{AB+C+}$

❏ 中⇒前:

● 加上完整的括號配對 (左結合): $((A+B)+C)$

● 將運算子取代最近的左括號: $((A+B)+C) \Rightarrow$



● 刪除右括號: $++ABC)) \Rightarrow \underline{++ABC}$



● Ex: $A \uparrow B \uparrow C$, 寫出其postfix及prefix

☒ 中⇒後:

● 加上完整的括號配對 (右結合): $(A \uparrow (B \uparrow C))$

● 將運算子取代最近的右括號: $(A \uparrow (B \uparrow C)) \Rightarrow$

● 刪除左括號: $(A(BC \uparrow \uparrow) \Rightarrow \underline{ABC \uparrow \uparrow}$

☒ 中⇒前:

● 加上完整的括號配對 (右結合): $(A \uparrow (B \uparrow C))$

● 將運算子取代最近的左括號: $(A \uparrow (B \uparrow C)) \Rightarrow$

● 刪除右括號: $\uparrow A \uparrow BC)) \Rightarrow \underline{\uparrow A \uparrow BC}$



※範例練習※

● Infix轉Postfix

❖ $(A+B \times C)-C \div (D \div E)$

(Ans: A B C \times + C D E $\div \div$ -)

❖ $\sim A \text{ and } B \text{ or } (C > E) \text{ and } \sim F$ (“ \sim ”表not)

(Ans: A \sim B and C E $>$ or F \sim and)

● Infix轉Prefix

❖ $A \uparrow -B + C \div (D - E)$

(Ans: + \uparrow A - B \div C - D E)

❖ $-B + A \uparrow 2 - 4 \times B - D \div E \uparrow F$

(Ans: --+ -B \uparrow A2 \times 4B \div D \uparrow EF)



後序式、前序式轉中序式

● Prefix \Rightarrow Infix

運算子 運算元 1 運算元 2 \Rightarrow 運算元 1 運算子 運算元 2

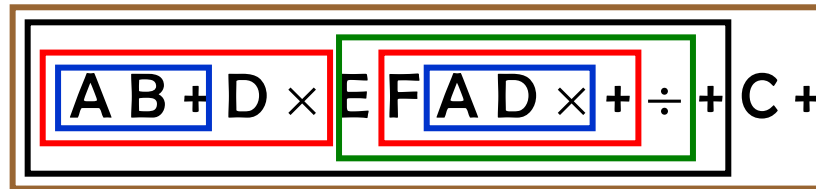
● Postfix \Rightarrow Infix

運算元 1 運算元 2 運算子 \Rightarrow 運算元 1 運算子 運算元 2



● Postfix: $AB+D\times EFAD\times+\div+C+$, 則其infix為何?

Sol:



$$\Rightarrow (((((A+B)\times D) + (E \div (F + (A\times D)))))) + C$$

$$\Rightarrow (A + B) \times D + E \div (F + A \times D) + C$$

Hint: 哪個括號能拿掉, 哪個不能拿掉, 自己要會判斷!!!

● Prefix: $+ - AB \div \times CD - EF$, 則其infix為何?

Sol:

$$+ \boxed{- A B} \div \boxed{\times C D} \boxed{- E F}$$

$$\Rightarrow ((A - B) - ((C \times D) \div (E - F)))$$

$$\Rightarrow A - B + C \times D \div (E - F)$$

Hint: 哪個括號能拿掉, 哪個不能拿掉, 自己要會判斷!!!



● Postfix: $AB-\uparrow CD\times+$, 則其infix為何?

Sol:

~~$AB-\uparrow CD\times+$~~

$A B - \uparrow C D \times +$

$\Rightarrow ((A \uparrow (-B)) + (C \times D))$

Hint: 哪個括號能拿掉, 哪個不能拿掉, 自己要會判斷!!!



※範例練習※

● Postfix: $6\ 2\ 3\ \times\ \div\ 4\ \times\ 5\ +$ 之值為何?

■ Ans: 9

● Postfix: $2\ 5\ 3\ 2\ 5\ -\ +\ \uparrow\ \times\ 5\ 3\ \times\ -$ 之值為何?

■ Ans: -13

● Postfix: $1\ 2\ \times\ 3\ -\ 4\ 5\ \times\ \times\ 6\ 7\ \times\ 8\ \div\ +$ 之值為何?

■ Ans: -14.75

● Postfix: $a\ b\ \div\ c\ -\ d\ e\ \times\ +\ a\ c\ \times\ -$, 則其Prefix為何?

■ Ans: $- + - \div a\ b\ c\ \times\ d\ e\ \times\ a\ c$

● Prefix: $\times\ +\ +\ +\ a\ b\ \times\ c\ +\ d\ e\ f\ +\ g\ h$, 則其Postfix為何?

■ Ans: $a\ b\ +\ c\ d\ e\ +\ \times\ +\ f\ +\ g\ h\ +\ \times$



中序式與後序式/前序式互轉的演算法

- 需要堆疊(Stack)支援。
- 討論次序:
 - ❖ Infix轉Postfix的演算法 (利用1個Stack)
 - ❖ Postfix之計算的演算法
 - ❖ Infix轉Prefix的演算法 (利用2個Stack)
 - ❖ Prefix之計算的演算法



Infix轉Postfix的演算法 (利用1個Stack)

● 演算法意義匯整:

1. 中序運算式由左往右掃描, 當遇到:

1-1. 運算元: 直接輸出 (或Print) 到後序式

1-2. 運算子:

1-2-1. “)” : pop堆疊內的運算子直到遇到 “(”

1-2-2. 其它運算子x: 比大小

1. 若運算子x的優先權 > 堆疊內最Top的運算子時, 則將運算子x push至堆疊中
2. 若運算子x的優先權 \leq 堆疊內最Top的運算子時, 則pop堆疊內的運算子直到x > 堆疊內最Top的運算子為止

2. 掃描完中序運算式, 則將堆疊內的殘餘資料pop完

● Note:

❏ Stack為空時, 其優先權最低。(∵ Stack沒有任何運算子可與待輸入的運算子做比較!!)

❏ “(” 在Stack外優先權最高, 但在Stack內優先權最低。



寫出 Infix: $A+B \times (C-D \div E)+F$ 轉 Postfix 的過程

- 1) Print 'A'
- 2) '+' > 空的Stack \Rightarrow push '+'
- 3) Print 'B'
- 4) '×' > '+' (堆疊的top元素) \Rightarrow push '×'
- 5) '(' > '×' \Rightarrow push '('
- 6) Print 'C'
- 7) '-' > '(' (on stack) \Rightarrow push '-'
- 8) Print 'D'
- 9) '÷' > '-' \Rightarrow push '÷'
- 10) Print 'E'
- 11) ')' \Rightarrow pop stack until '('
 - 1) Pop '÷', print
 - 2) Pop '-', print
 - 3) Pop '(', 不用print
- 12) '+' \leq '×' \Rightarrow pop '×', print
- 13) '+' \leq '+' \Rightarrow pop '+', print
- 14) '+' > 空的Stack \Rightarrow push '+'
- 15) Print 'F'
- 16) Scan完畢, 清空Stack \Rightarrow pop '+'

Ans:

A B C D E ÷ - × + F +

÷
-
(
×
+



- 在執行上述轉換過程中，所需之Stack Size至少需要 \geq __個儲存空間



```
while (Infix尚未Scan完畢) do    //意義匯整 1.
begin
  x = NextToken; //Token是指運算的單元, 可能是運算元或是運算子
  if (x是operand) //意義匯整 1-1.
    print(x);
  else
    begin //意義匯整 1-2.
      if (x是 '(') //意義匯整 1-2-1.
        begin
          pop(S), 直到遇見 '(' 為止;
        end;
      else begin //意義匯整 1-2-2.
        比較 (x與stack top之優先權)
        - case "x > top": 則push(x, S);
        - case "x ≤ top": 則pop(S), 直到x > top為止, 再push(x, S);
      end;
    end;
end;
while (stack ≠ empty) do    //意義匯整 2.
  pop stack;
```



Postfix之計算的演算法

43-15 \times +, 寫出postfix計算過程 (含Stack內容)。

Stack容量需至少為多少?

1) Push '4'

2) Push '3'

3) '-':

1) pop '3'與 '4'

2) 計算 $4-3 = 1$, 再push '1'

4) Push '1'

5) Push '5'

6) ' \times ':

1) pop '5'與 '1'

2) 計算 $1 \times 5 = 5$, 再push '5'

7) '+':

1) pop '5'與 '1'

2) 計算 $1 + 5 = 6$, 再push '6'

8) Scan完畢, pop stack $\Rightarrow 6$

5
5
6

Ans:



while (Infix尚未Scan完畢) do

begin

x = NextToken; //Token是指運算的單元, 可能是運算元或是運算子

if (x是operand)

push(x);

else

begin

pop適當數目的運算元

- 計算

- 再將計算結果**push**回**stack**

end;

end;

pop stack; //即為結果



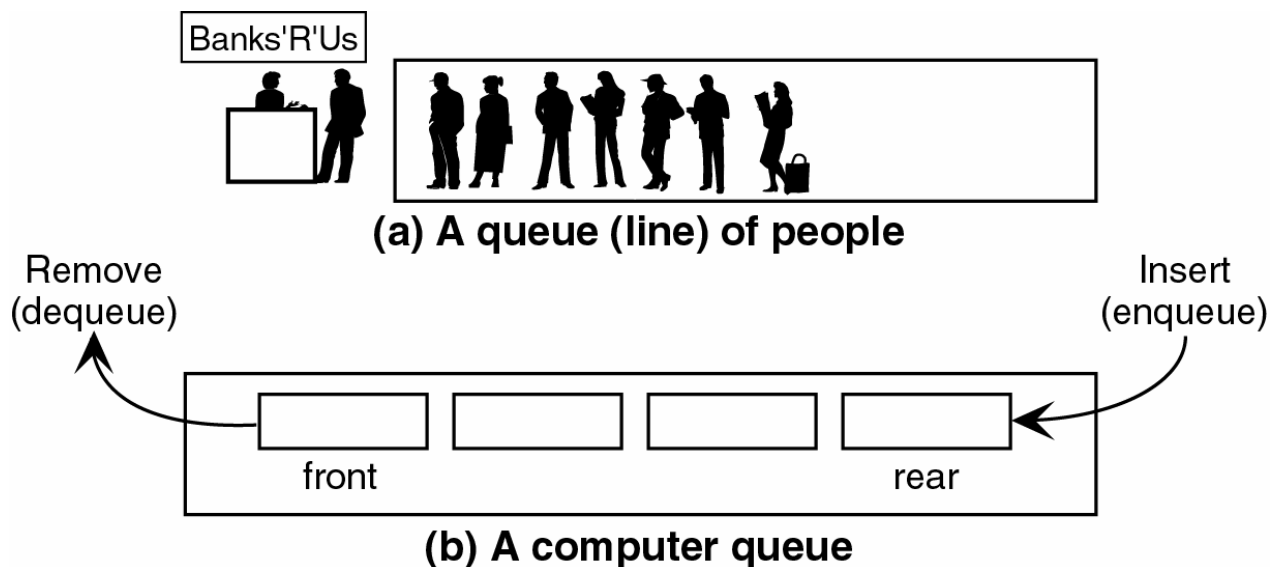
※範例練習※

- Postfix: $526 \times + 42 \div 3 \times -$ 之求算過程 (含Stack內容), 且 Stack Size 最少需為多少?
 - 結果 = 11, Stack Size ≥ 3
- 在求算Postfix: $AB \div C - DE \times + AC \times -$ 時, Stack Size 最少需為多少?
 - Stack Size ≥ 3

Queue (佇列)

Def:

- 具有FIFO (First-in, First-out) 性質的有序串列
- 插入與刪除元素的動作發生在佇列的不同端
 - 插入動作發生在尾端 (Rear)
 - 刪除動作發生在前端 (Front)



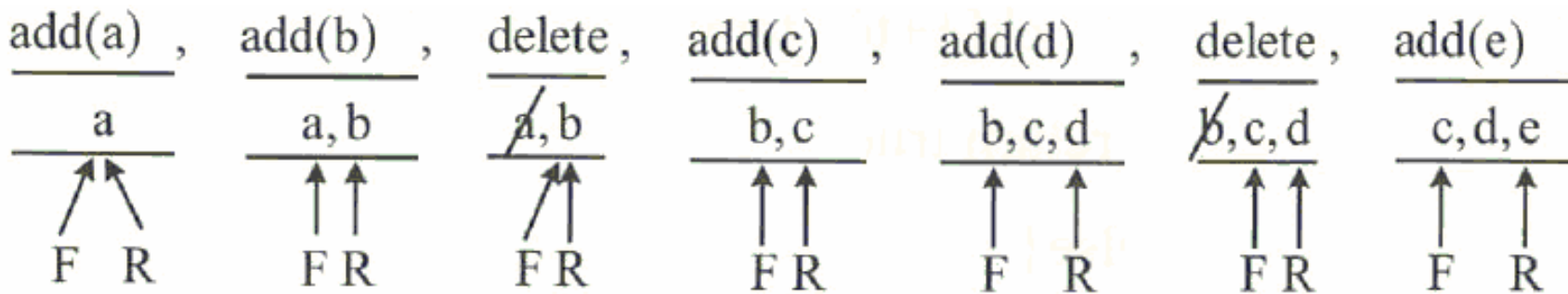


Queue Example

- 有一空的queue, 實施下列動作後, 則Queue的內容為何?

add(Q, a), add(Q, b), delete, add(Q, c), delete, add(Q, e)

Ans:



F: front, R:rear



Queue的應用

- 日常生活的排隊行為。
- 在作業系統中的job scheduling, 在相同的priority下, 利用queue來完成先到先作的策略。
- 有許多的I/O工作同時要處理。將所有的I/O要求, 利用queue來達成先到先作的策略。
- 用於模擬 (Simulation) 方面, 如佇列理論 (Queuing Theory).
 - ❖ The two factors that most affect the performance of queues are the **arrival rate** and the **service time**.



Queue的ADT

Data Objects:

- ❖ Queue: a set of data items
- ❖ Front: 指示Queue之前端元素所在
- ❖ Rear: 指示Queue之尾端元素所在

Operations:

- ❖ Create(Q): 建立空佇列Q
- ❖ ADDQ(Q, item) \Rightarrow Q: 將item插入到Queue Q中, 成為新的尾端元素
(if Queue is full, then 無法執行)
- ❖ DeleteQ(Q, item) \Rightarrow item, Q: 刪除Queue中的前端元素 (if Queue is empty, then 無法執行)
- ❖ IsEmpty(Q) \Rightarrow Boolean
- ❖ IsFull(Q) \Rightarrow Boolean
- ❖ Front(Q) \Rightarrow item: 傳回Queue之Front端元素 (但不刪除)



Queue的製作

● 用Link list製作

- Single link list

● 用Array製作

- 利用Linear Array
- 利用Circular Array with $(n-1)$ space used
- 利用Circular Array with n space used



■ 用 Linked list 製作

● Create(Q)

■ 宣告:

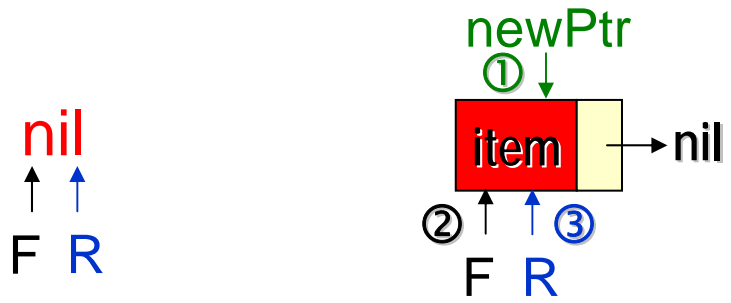
● rear: pointer = nil

● front: pointer = nil

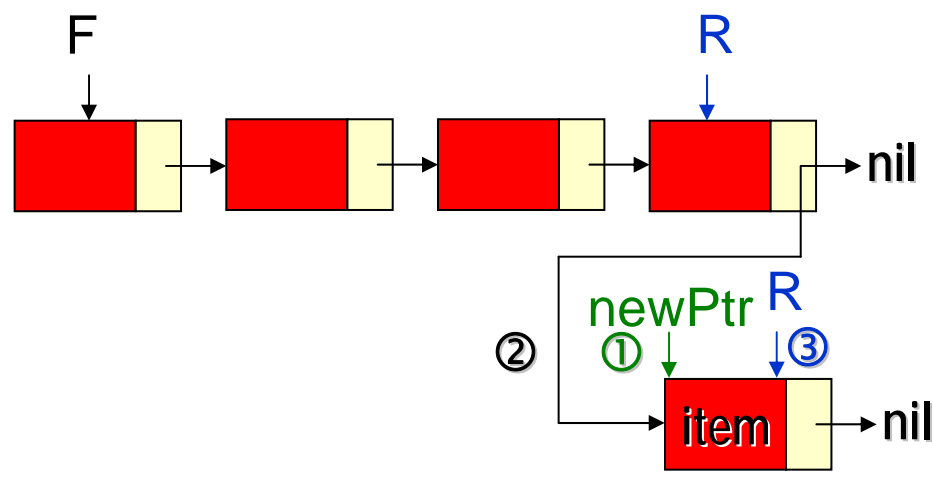


● **ADDQ(Q, item)** //為說明方便起見, 下列的 F = front, R = rear

❖ Case 1: (當Queue為空佇列)



❖ Case 2: (當Queue不為空佇列)





//F = front, R = rear

begin

New(newPtr);

newPtr → data = item;

newPtr → link = nil;

if (rear = nil) then //Case 1

front = newPtr;

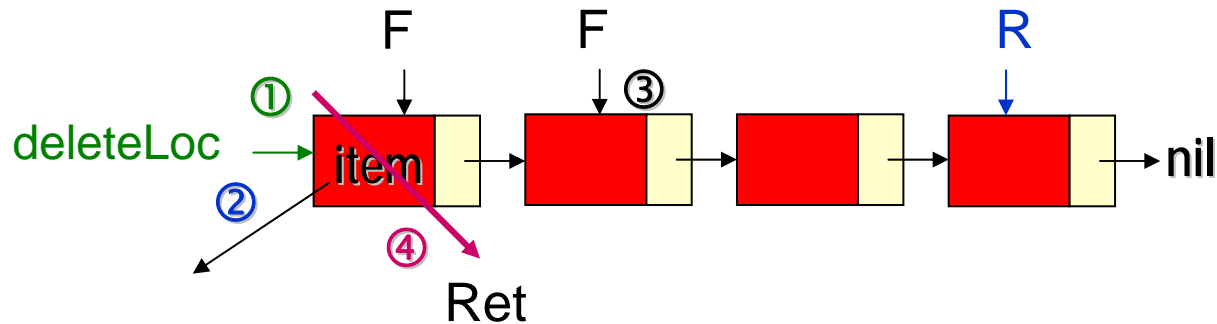
else // Case 2

rear → link = newPtr;

rear = newPtr;

end

Delete(Q)



begin

if (front = nil) then

Queue Empty;

else begin

① **deleteLoc = front;**

② **item = front → data;**

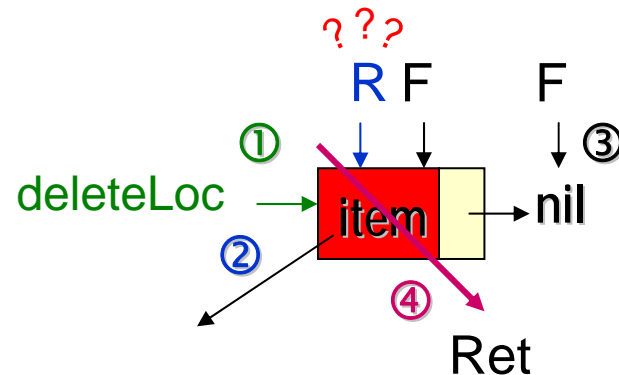
③ **front = front → link;**

④ **Ret(deleteLoc);**

if (front = nil) then
rear = nil;

end;

end



- 假設Queue中只有一個node, 回收後把Rear指向nil.
- 主要是耽心系統不會自動將Rear設成nil, 使得Rear指標無效!!



用 Array 製作

- 利用 Linear Array
- 利用 Circular Array with **(n-1) space** used
- 利用 Circular Array with **n space** used



利用 Linear Array

Create(Q)

宣告:

Q: array[0...n-1] of items //宣告Q是一個大小為n的一維Array

Front: integer = -1 //初值

Rear: integer = -1 //初值

AddQ(Q, item) ⇒ Queue

begin

if (rear = n) then

QueueFull;

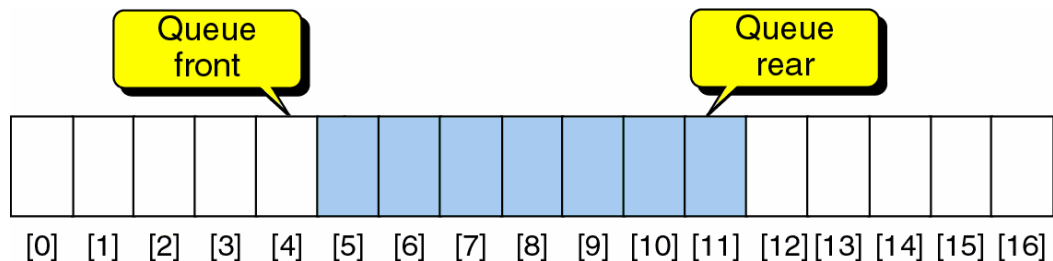
else **begin**

rear = rear + 1

Q[rear] = item

end

end.



● DeleteQ(Q) \Rightarrow item, Queue

begin

if (rear = front) then

QueueEmpty;

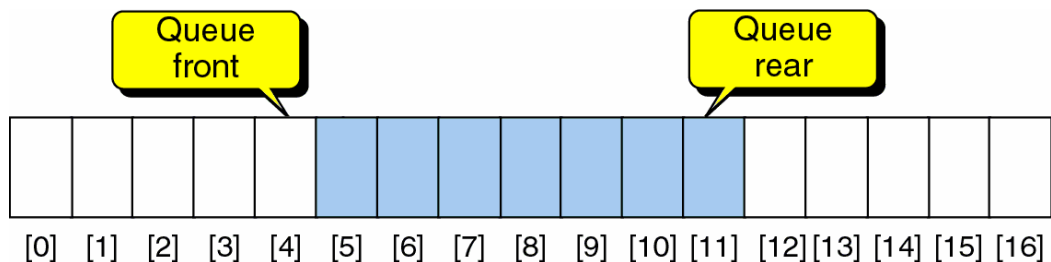
else **begin**

front = front + 1

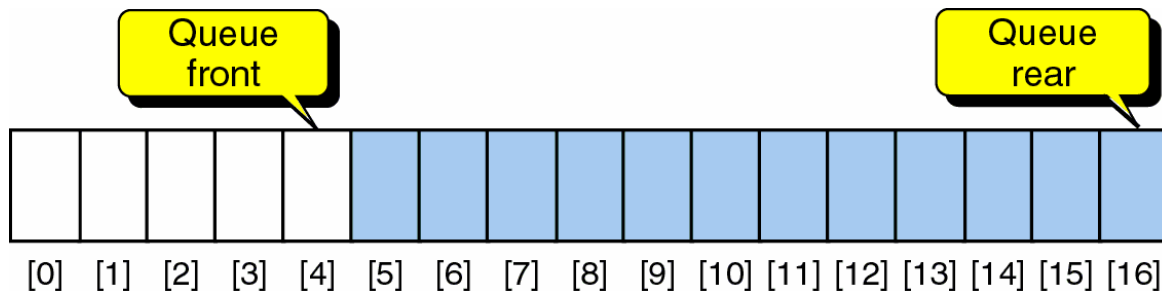
item = Q[front]

end

end.



● 問題: 當 rear = n 時, Queue並不代表真正為滿的情況!



- 為解決上述問題，我們或許可以設計一個副程式，當資料已成長到Arrar的最末端時，作一次“**是否真的為滿**”的判斷（即：**Rear = n** 且 **Front = 0**）。若不為真滿，則需將 (Front+1) 到Rear端的所有元素往左移Front格，並重設Rear與Front的指標值。
- 然而，此種作法會導致Queue之Add動作時間為 $O(n)$
 - ❏ \therefore 是用**迴圈**來實作資料的搬移，花費時間太大。同時，此搬移工作是額外的處理項目，與Add動作本身是無關的。
 - ❏ \therefore 當Add的工作很頻繁時，整體執行效益差



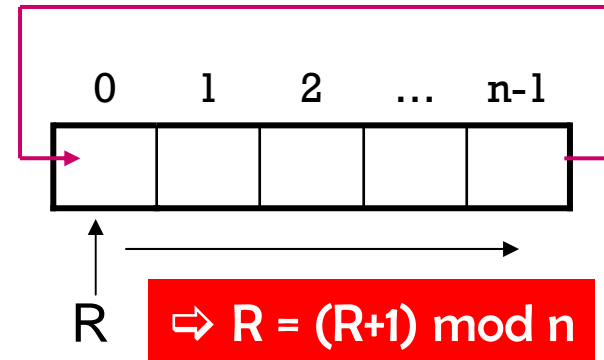
利用 Circular Array with (n-1) space used

● Create(Q)

■ 宣告:

● $Q: \text{Array}[0 \dots n-1]$

● $\text{front} = \text{rear} = 0$ //初值



● AddQ(item, Q) \Rightarrow Queue

begin

$\text{rear} = (\text{rear} + 1) \bmod n;$ //rear指標先前進

if $\text{rear} = \text{front}$

 QueueFull; //表示Queue滿了

$\text{rear} = \text{rear} - 1 \bmod n;$ //將rear重設回前一格

else

$Q[\text{rear}] = \text{item};$

end;



DeleteQ(Q) \Rightarrow item

begin

if **front=rear** //先檢查

QueueEmpty;

else begin

front = (front+1) mod n;

item = Q[front];

end;

end;

特點:

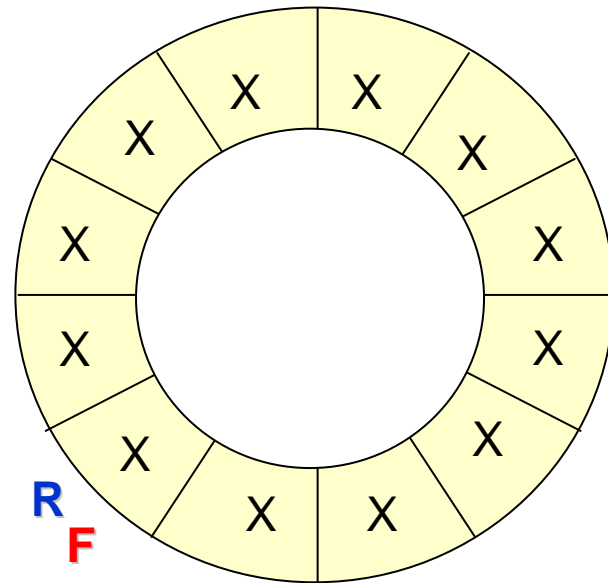
❑ 最多只利用到 **n-1 格空間**

❑ 若硬要使用到 n 格空間, 則 rear = front 條件成立時, **無法真正區分出 Queue為Full或Empty**

● \therefore 判斷Full與Empty的條件式相同 (皆為**rear = Full**)

❑ Add與Delete之動作時間皆為**O(1)**

● \therefore **沒有資料挪移**的動作!!





利用 Circular Array with n space used

- 引進一個 **Tag** 變數, 用以 **協助判斷** Queue 為 Empty 或 Full:
 - ❑ 該變數為 Boolean 型態
 - ❑ 若 Tag = True: 則可 協助判斷 是否為 Full
 - ❑ 若 Tag = False: 則可 協助判斷 是否為 Null
 - ❑ **不是光靠 Tag 就能做正確判斷!!**



● Create(Q)

■ 宣告:

● Q: Array[0...n-1]

● front = rear: int = 0 //初值

● Tag: Boolean = 0 //初值

● AddQ(item, Q) \Rightarrow Queue

begin

if (rear = front and Tag = 1)

QueueFull;

else begin

rear = (rear+1) mod n; //rear指標前進

Q[rear]=item;

if (rear=front)

Tag=1;

end;

end;

DeleteQ(Q) \Rightarrow item

begin

if (Front=Rear and Tag=0)

QueueEmpty;

else begin

Front = (Front+1) mod n;

item = Q[Front];

if (Front=Rear)

Tag=0;

end;

end;

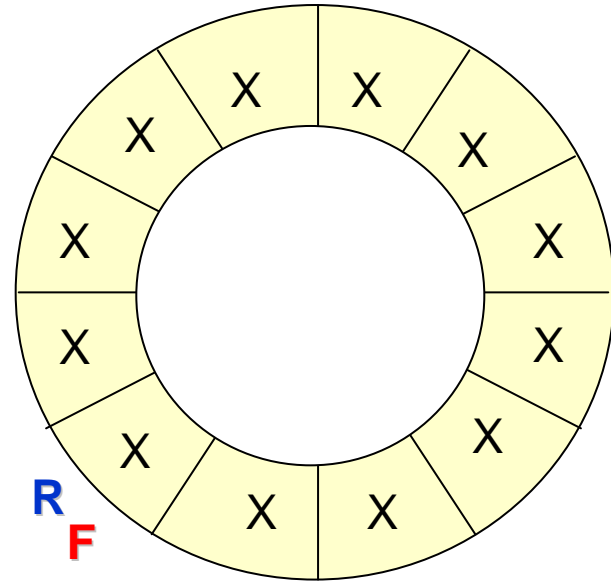
特點:

❖ 最多可利用到 **n** 格空間

❖ Add與Delete之運作時間稍長

● \therefore 多了一條**if測試**, 來測試Tag值設定, 且此兩個運作使用上極頻繁

● \therefore **整體時間效益稍嫌Poor!!**





Queue的種類

- FIFO Queue (先進先出佇列)
- Priority Queue (優先權佇列)
- Double Ended Queue (雙邊佇列)
- Double Ended Priority Queue (雙邊優先佇列)



● **FIFO Queue (先進先出佇列)**

■ 即一般的佇列，具有FIFO特性，前端刪除元素，尾端加入元素

● Priority Queue (優先權佇列)

● Double Ended Queue (雙邊佇列)

● Double Ended Priority Queue (雙邊優先佇列)



● FIFO Queue (先進先出佇列)

● **Priority Queue (優先權佇列)**

- ❏ 不一定遵守FIFO特性

- ❏ 運作：

 - 插入任意優先權值之元素

 - 刪除時，是刪除具最大/最小優先權值之元素

- ❏ 可利用Heap (堆積) 來製作。

● Double Ended Queue (雙邊佇列)

● Double Ended Priority Queue (雙邊優先佇列)



- FIFO Queue (先進先出佇列)
- Priority Queue (優先權佇列)
- **Double Ended Queue (雙邊佇列)**
 - 可於任何一端執行插入/刪除元素的動作
 - 亦可實作成：
 - Input-restricted: 插入動作在固定端, 刪除動作在任意端
 - Output-restricted: 插入動作在任意端, 刪除動作在固定端
- Double Ended Priority Queue (雙邊優先佇列)



- FIFO Queue (先進先出佇列)
- Priority Queue (優先權佇列)
- Double Ended Queue (雙邊佇列)
- **Double Ended Priority Queue (雙邊優先佇列)**
 - 可於任何一端執行插入元素的動作。但刪除時，有一端是做Delete Max元素的動作，另一端則作Delete Min元素的動作。
- 可利用Min-Max Heap (堆積) 來製作。



補充



Prefix之計算

- 觀念同Postfix的計算過程，都是利用一個Stack。
- 差別：
 - ❖ 由右往左Scan
 - ❖ Operand在pop之後的計算位置相反



計算 $- \times + 123 \div 84$

1) Push '4'

2) Push '8'

3) '÷':

1) pop '8'與 '4'

2) 計算 $8 \div 4 = 2$, 再push '2'

4) Push '3'

5) Push '2'

6) Push '1'

7) '+':

1) pop '1'與 '2'

2) 計算 $1 + 2 = 3$, 再push '3'

8) '×':

1) pop '3'與 '3'

2) 計算 $3 \times 3 = 9$, 再push '9'

9) '-':

1) pop '9'與 '2'

2) 計算 $9 - 2 = 7$, 再push '7'

10) Scan完畢, pop stack $\Rightarrow 7$

Ans:

1
3
9
7



■ Infix轉Prefix之演算法

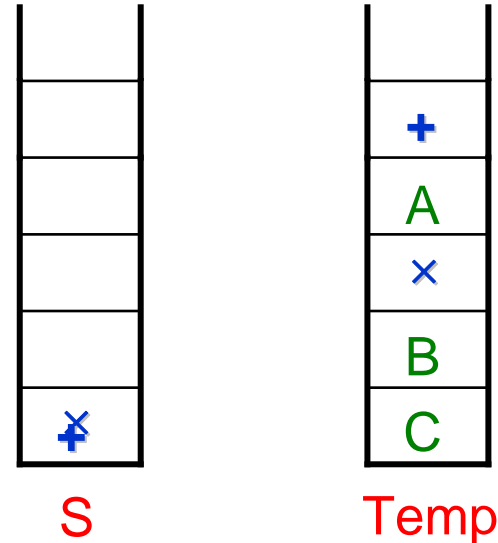
● 大原則:

- ❖ Infix由右往左Scan
- ❖ 需要2個Stacks支援



寫出 Infix: $A+B\times C$ 轉 Prefix 的過程

- 1) push 'C' 至 Temp Stack
- 2) ' \times ' > 空的Stack S \Rightarrow push ' \times '
- 3) push 'B' 至 Temp Stack
- 4) ' $+$ ' \leq ' \times ' (堆疊的top元素)
 - 1) pop ' \times ' \Rightarrow push into Temp Stack
 - 2) Push ' $+$ ' into stack S
- 5) push 'A' 至 Temp Stack
- 6) Scan完畢, 清空Stack S \Rightarrow pop ' $+$ ' and push into Temp Stack
- 7) Pop Temp Stack內所有資料

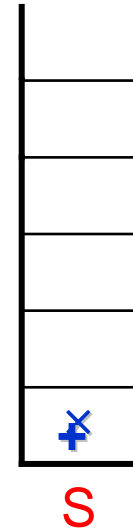


Ans: $+ A \times B C$



上述過程若沒有Temp Stack會發生何事?

- 1) print 'C'
- 2) '×' > 空的Stack S \Rightarrow push '×'
- 3) print 'B'
- 4) '+' \leq '×' (堆疊的top元素)
 - 1) pop '×' \Rightarrow print '×'
 - 2) Push '+' into stack S
- 5) print 'A'
- 6) Scan完畢, 清空Stack S \Rightarrow pop '+' and print it



Ans: C B × A +

由此可知, Temp Stack扮演“反向輸出”的角色